HASKELL USEFUL FUNCTION

span:: (a -> Bool) -> [a] -> ([a], [a])

span, applied to a predicate p and a list xs, returns a tuple of xs of elements that satisfy p and second element is the remainder of the list: > span (< 3) [1,2,3,4,1,2,3,4] == ([1,2],[3,4,1,2,3,4]) > span (< 9) [1,2,3] == ([1,2,3],[]) > span (< 0) [1,2,3] == ([],[1,2,3]) span p xs is equivalent to (takeWhile p xs, dropWhile p xs)

take:: Int -> [a] -> [a]

take n, applied to a list xs, returns the prefix of xs of length n, or xs itself if n > length xs: > take 5 "Hello World!" == "Hello" > take 3 [1,2,3,4,5] == [1,2,3] > take 3 [1,2] == [1,2] > take 3 [] == [] > take (-1) [1,2] == [] > take 0 [1,2] == [] It is an instance of the more general Data.List.genericTake, in which n may be of any integral type.

takeWhile:: (a -> Bool) -> [a] -> [a]

takeWhile, applied to a predicate p and a list xs, returns the longest prefix (possibly empty) of xs of elements that satisfy p: > takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2] > takeWhile (< 9) [1,2,3] == [1,2,3] > takeWhile (< 0) [1,2,3] == []

drop :: Int -> [a] -> [a]

drop n xs returns the suffix of xs after the first n elements, or [] if n > length xs: > drop 6 "Hello World!" == "World!" > drop 3 [1,2,3,4,5] == [4,5] > drop 3 [1,2] == [] > drop 3 [] == [] > drop (-1) [1,2] == [1,2] > drop 0 [1,2] == [1,2] It is an instance of the more general Data.List.genericDrop, in which n may be of any integral type.

dropWhile:: (a -> Bool) -> [a] -> [a]

dropWhile p xs returns the suffix remaining after takeWhile p xs: > dropWhile (< 3) [1,2,3,4,5,1,2,3] == [3,4,5,1,2,3] > dropWhile (< 9) [1,2,3] == [] > dropWhile (< 0) [1,2,3] == [1,2,3]

concatMap:: (a -> [b]) -> [a] -> [b]

Map a function over a list and concatenate the results.

replicate:: Int -> a -> [a]

replicate n x is a list of length n with x the value of every element. It is an instance of the more general Data.List.genericReplicate, in which n may be of any integral type.

(!!):: [a] -> Int -> a

List index (subscript) operator, starting from 0. It is an instance of the more general Data.List.genericIndex, which takes an index of any integral type.

splitAt:: Int -> [a] -> ([a], [a])

splitAt n xs returns a tuple xs prefix of length n and second element is the remainder of the list: > splitAt 6 "Hello World!" == ("Hello ","World!") > splitAt 3 [1,2,3,4,5] == ([1,2,3],[4,5]) > splitAt 1 [1,2,3] == ([1],[2,3]) > splitAt 3 [1,2,3] == ([1,2,3],[]) > splitAt 4 [1,2,3] == ([1,2,3],[]) > splitAt 0 [1,2,3] == ([],[1,2,3]) > splitAt (-1) [1,2,3] == ([],[1,2,3]) It is equivalent to (take n xs, drop n xs). splitAt is an instance of the more general Data.List.genericSplitAt, in which n may be of any integral type.

flip:: (a -> b -> c) -> b -> a -> c

it evaluates the function flipping the order of the arguments; flip (/) 1 2 == 2.0

cycle :: [a] -> [a]

cycle ties a finite list into a circular one, or equivalently, the infinite repetition of the original list. It is the identity on infinite lists.
cycle “abc” == abcabcabcabcabcabcabcabcabcabcabcabcabc...

foldl:: (a -> b -> a) -> a -> [b] -> a

foldl, applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right: > foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn The list must be finite.

foldr:: (a -> b -> b) -> b -> [a] -> b

foldr, applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left: > foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)